

Recent work with RAJA, and a nested loop update

DOE Centers of Excellence Performance Portability
Meeting 2017

Adam J. Kunen

August 23, 2017



RAJA is a C++ abstraction layer that enables portability with small disruption to application programming styles

The main goal of RAJA is to balance performance...

- Preserve and augment abilities of C++ compilers to optimize
- Support various forms of fine-grained (on-node) parallelism and various programming model options (OpenMP, CUDA, TBB, OpenACC, ...)

... and developer productivity

- Maintain single-source kernels and don't bind an app to a particular PM
- Clear separation of responsibilities
 - **RAJA:** Execute loops, encapsulate hardware & programming model details
 - **Application:** Select loop iteration patterns and execution policies with RAJA API

RAJA development is currently driven by the needs of ATDM/ASC applications at LLNL and ECP collaborators

RAJA concepts help encapsulate loop execution details

C-style for-loop

```
double* x ; double* y ;
double a, tsum = 0, tmin = MYMAX;

for ( int i = begin; i < end; ++i ) {
    y[i] += a * x[i] ;
    tsum += y[i] ;
    if ( y[i] < tmin ) tmin = y[i];
}
```

RAJA-style loop

```
double* x ; double* y ;
double a ;
RAJA::SumReduction<reduce_policy, double> tsum(0);
RAJA::MinReduction<reduce_policy, double> tmin(MYMAX);

RAJA::forall< exec_policy > ( IndexSet , [=] (int i) {
    y[i] += a * x[i] ;
    tsum += y[i];
    tmin.min( y[i] );
} );
```

- **RAJA decouples loop iteration and loop body**
 - Iterations are “tasks” – aggregate, reorder, etc.
- **RAJA Concepts:**
 - **Patterns:** forall, forallN, reduce, scan
 - **Policies:** sequential, simd, openmp, cuda,
 - **Index:** iterations – aggregate, reorder, tile,

Execution patterns & policies
(scheduling, PM choice, etc.)

IndexSets
(iteration space, ordering, etc.)

Portable Reduction types

Loop body is mostly unchanged (C++ lambda function).

Why we prefer RAJA over alternatives

- “Light touch”
 - Existing application data structures & algorithms require little change, if any
- “Low barrier to entry”
 - Parallelism can be added selectively and performance tuned incrementally
- “Application-facing design philosophy”
 - Maps naturally to apps and can be customized – easy to grasp for (non-CS) application developers
- “Performance”
 - RAJA does well with “streaming” kernels that are prevalent in LLNL codes
 - Designed for coarse-grained synchronization – reduces resource contention and memory synchronization overheads

RAJA developments since last year and WIP


- Cleaner organization of concepts & header files, refined APIs
- Backends for OpenMP4.x, OpenACC, TBB
- Parallel scans
- New IndexSet implementation supports arbitrary segment types
- “Multi-policy” for runtime policy selection
- Improved integration with CHAI
- RAJA Performance Suite – run various experiments to compare kernels (RAJA vs. native), help guide compiler NRE work
- Expanded and refined nested loop capabilities and API

Nested loop roadblocks


- Recent work with NVidia nvcc, IBM hackathon at LLNL
 - Identified performance issues in nested-loop abstractions (RAJA::forallN)
 - Copy construction of loop body
 - Capture-by-value vs. capture-by-reference causing issues with nvcc correctness
- Rework of forallN
 - Current implementation of forallN relies on a “peel and bind” mechanism to generate the nested loop structure and bind the loop iterates to the lambda
 - Causes excessive copy construction – seen as massive performance problem with things like CHAI, host-device lambdas, reduction object.
 - Revamp of forallN replaces “peel and bind” with “peel and set” mechanism that doesn’t trigger copy construction

Current Peel-and-Bind implementation of RAJA::forallN

```
For i : I  
  For j : J  
    For k : K  
      body(i,j,k)
```



```
auto body = [=](int i, int j, int k){ ... };  
RAJA::forallN<pol>(I, J, K, body);
```



- Each loop performs two **capture-by-value** wrappings of the loop body
 - One peels off that loops execution policy and segment
 - The other binds that loops iterate to the body (similar to std::bind)
 - Number of copy-constructions of body $O(I*J*K)$

```
A = IndexConverter(body)  
B = PeelOuter(A)  
For i : I  
  C = BindFirstArg(A, i)  
  D = PeelOuter(C)  
  
  For j : J  
    E = BindFirstArg(C, j)  
    F = PeelOuter(E)  
  
    For k : K  
      G = BindFirstArg(E, k)  
      G()
```



Why was Peel-and-Bind used if it's so inefficient?!?

- It was straightforward to design
 - An initial implementation
- We just didn't know
 - Often the "body" is a lambda which only captures POD types
 - The compiler can eliminate most of the copy constructors and inline everything
 - There is no apparent inefficiency
- So what happened?
 - Three things:
 - We used RAJA reduction objects
 - We used CHAI
 - CUDA host-device lambdas
 - These have explicit copy constructors
 - The compiler does not optimize these away
 - Performance drops through the floor


We only see performance loss when our lambdas capture complex objects

Reengineered nested loop execution invokes the loop body with a tuple of indices

```
For i : I  
  For j : J  
    For k : K  
      body(i,j,k)
```



```
auto body = [=](int i, int j, int k){ ... };  
RAJA::forallN<pol>(I, J, K, body);
```



- Each loop assigns its iterate into a tuple
 - One wrapping of body is needed to provide invocation
 - Wrapper can be captured-by-reference at each loop nest level
 - Number of loop-body copy constructions is $O(1)$
 - Side Benefit: New portable metaprogramming library “camp”

```
idx = std::tuple<int, int, int>  
A = InvokeWrapper(body)  
For i : I  
  idx.i = i  
  
  For j : J  
    idx.j = j  
  
    For k : K  
      idx.k = k  
  
      A(idx)
```

Conclusion

- A lot of things are going on in RAJA
 - New features
 - New backends
- Running up against performance portability issues with CUDA and OpenMP 4.5 that are forcing us to rethink certain implementation strategies
- Bug reports, feature requests, code contributions, are all welcome!
- Get RAJA on github:
 - <https://github.com/LLNL/RAJA>



Number of loop-body copy constructions for the Peel-and-Bind implementation

$$\text{Copies}(I_0) = 2 + \|I_0\|$$

$$\text{Copies}(I_0, I_1) = 2 + 2\|I_0\| + \|I_0 \times I_1\|$$

$$\text{Copies}(I_0, I_1, I_2) = 2 + 2\|I_0\| + 2\|I_0 \times I_1\| + \|I_0 \times I_1 \times I_2\|$$

$$\begin{aligned}\text{Copies}(\{I_i\}) &= 2 + \left(\sum_{i=1}^N 2 \prod_{j=1}^i \|I_j\| \right) + \prod_{j=1}^N \|I_j\| \\ &= \mathcal{O} \left(\prod_{j=1}^N \|I_j\| \right)\end{aligned}$$

The number of copy-ctors called is on the order of the iteration space size